Crowfoot: a verifier for higher-order store programs*

Nathaniel Charlton Ben Horsfall Bernhard Reus
{n.a.charlton,b.g.horsfall,bernhard}@sussex.ac.uk

Department of Informatics, University of Sussex

Abstract. We present Crowfoot, an automatic verification tool for imperative programs that manipulate procedures dynamically at runtime; these programs use a heap that can store not only data but also code (commands or procedures). Such heaps are often called *higher-order store*, and allow for instance the creation of new recursions on the fly. One can use higher-order store to model phenomena such as runtime loading and unloading of code, runtime update of code and runtime code generation. Crowfoot's assertion language, based on separation logic, features *nested Hoare triples* which describe the behaviour of procedures stored on the heap. The tool addresses complex issues like deep frame rules and recursion through the store, and is the first verification tool based on recent developments in the mathematical foundations of Hoare logics with nested triples.

1 Introduction

Dynamic memory that can store not only data but also code is often called *higher-order store*. Such memory thus allows program code to change during execution with the manipulation performed by the program itself. For instance, one may be able to write code onto a mutable heap, invoke it, manipulate it, and then invoke it again later when needed. With higher-order store one can model phenomena such as runtime loading and unloading of code — as performed in plugin systems, operating system kernels and dynamic software update systems — and runtime code generation.

Logics with *nested triples* [17, 11], where assertions can contain Hoare triples which describe the behaviour of code stored on the program's heap, have been proposed as a way to reason modularly about higher-order store programs. Recent developments [17, 18] have provided solid theoretical foundations for separation logics with nested triples. In this paper we present Crowfoot, the first automatic verification system to apply these developments in practice. Crowfoot has been inspired by previous tools for automated verification using (conventional) separation logic, such as Smallfoot [3].

* We acknowledge the support of EPSRC grant *"From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs"* (EP/G003173/1).

The Crowfoot tool provides (semi-)automatic verification for imperative programs which make use of higher-order store. Crowfoot uses an extension of separation logic for higher-order store, and performs its proofs by symbolic execution [4]. The main distinctive features of the Crowfoot verifier are:

- availability of nested triples for reasoning about stored procedures
- built-in support for recursive specifications for recursion through the store
- built-in support of the "deep frame rule", allowing correct and powerful framing of invariants in the presence of stored procedures
- built-in support of partial application of stored procedures
- an automatic prover for entailments between triples (as well as the usual entailments between assertions), supporting modular verification
- a sound theoretical underpinning of the implementation

Running example. We demonstrate Crowfoot using the program in Fig. 1. Note that grey shaded parts are annotations for the verifier and are *not* part of the program code. They will be explained in Section 2.3. Our example concerns a recursive implementation *fib* of the Fibonacci function, which makes its recursive calls through the store. Since the "internal" recursive calls are made through the store, we can "hook into" the recursion and provide a memoisation routine *mem* which also caches these internal calls. This kind of memoisation cannot be implemented for a conventional recursive implementation of the Fibonacci function. This is more challenging than the factorial function which is typically used [11, 2, 9] to illustrate recursion through the store.

We will use Crowfoot to prove that the *fib* code, with or without the memoiser, is memory safe and correctly computes the Fibonacci function. In the process we will demonstrate the features of Crowfoot which make this possible.

2 Programming and assertion languages

2.1 Programming language featuring higher-order store

Crowfoot works with an imperative heap-manipulating language with recursive procedures and, crucially, higher-order store operations. Fig. 2 includes a grammar for program statements. Square brackets are used for dereferencing addresses, so $x := [a]$ *reads* the content at address a into the variable x, whereas $[a] := x$ *stores* the value of x at address a in the heap[1].

There are two statements for using the higher-order store. Statements like $[a] :=$ proc $\mathcal{F}(x, \_)$ write the code of fixed procedure $\mathcal{F}$ to the heap at address a. Each argument is either a variable or the _ symbol; where variables are given these are used to perform *partial application* of the procedure. Allowing procedures to be partially applied at the time they are stored on the heap is the

[1] In Fig. 2, where there is a danger of confusion, we write `[]` for square brackets that are part of the programming language, and [] for "meta-brackets" that are used in grammar definitions. We write | for choice and ? for optional elements.

```
const res;

proc fib(a, n) {
 locals p, q, k;
 if n ≤ 0 then {
  [res] := 0;
  ghost fold $Rel(n, 0)
 } else {
  if n = 1 then {
   [res] := 1;
   ghost fold $Rel(?, ?)
  } else {
   k := n − 2;
   eval [a](a, k); p := [res];
   k := n − 1;
   eval [a](a, k); q := [res];
   [res] := p + q;
   ghost fold $Rel(n, p+q)
  }}}

proc mem(lookupL, addL, createL,
    disposeL, al, f, a, n) {
 locals found, b, v;
 ghost unfold $S(?, ?, ?, ?, ?, ?, ?);
 found := new 0;
 eval [lookupL](al, n, found, res);
 b := [found]; dispose found;
 if b = 0 then {
  ghost fold $S(?, ?, ?, ?, ?, ?, ?);
  eval [f](a, n);
  ghost unfold $S(?, ?, ?, ?, ?, ?, ?);
  v := [res]; eval [addL](al, n, v)
 } else { skip };
 ghost fold $S(?, ?, ?, ?, ?, ?, ?) }

proc useFib(lookupL, addL, createL,
    disposeL) {
 locals al, a, f, n;
 f := new 0;
 al := new 0;
 eval [createL](al);
 [f] := proc fib(_, _) deepframe DeepInv;
 a := new 0;
 [a] := proc mem(lookupL, addL, createL,
    disposeL, al, f, _, _);
 ghost fold $S(?, ?, ?, ?, ?, ?, ?);
 n := 31337;
 eval[a](a, n);
 ghost unfold $S(?, ?, ?, ?, ?, ?, ?);
 ghost unfold $ListLibWeak(?, ?, ?, ?);
 eval [disposeL](al);
 dispose a; dispose f; dispose lookupL;
 dispose addL; dispose createL;
 dispose disposeL; dispose res
}

proc main() {
 locals lookupL, addL, createL, disposeL;
 lookupL := new 0; addL := new 0;
 createL := new 0; disposeL := new 0;
 call load_list_lib(lookupL, addL,
    createL, disposeL);
 ghost unfold $ListLibStrong(?, ?, ?, ?);
 ghost fold $ListLibWeak(?, ?, ?, ?);
 call useFib(lookupL, addL, createL, disposeL)
}

proc load_list_lib(lookupL,
 addL, createL, disposeL) {...}
```

Fig. 1. Our running example program. (*DeepInv* is defined in Fig. 4.)

integer variables x, fixed procedure names $\mathcal{F}$, integer literals n, declared constants c

$$
\begin{array}{rcl}
\text{address expr } e_A & ::= & x \mid c \mid x + n \mid x + c \\
\text{value expr } e_V & ::= & n \mid x \mid c \mid e_V + e_V \mid e_V - e_V \mid e_V \times e_V \\
\text{statement } C & ::= & \mathsf{skip} \mid At \mid C; C \mid \mathsf{if}\ e_V = e_V\ \mathsf{then}\ C\ \mathsf{else}\ C \\
 & & \mid \mathsf{while}\ e_V = e_V\ \mathsf{do}\ C \mid \mathsf{while}\ e_V \neq e_V\ \mathsf{do}\ C \\
\text{argument } t & ::= & x \mid c \\
\text{atomic statement } At & ::= & x := e_V \mid x := [e_A] \mid [e_A] := e_V \mid [e_A] := [e_A] \\
 & & \mid x := \mathsf{new}\ {e_V}^+ \mid \mathsf{dispose}\ e_A \mid \mathsf{call}\ \mathcal{F}(t^*) \\
 & & \mid \mathsf{eval}\ [e_A](t^*) \mid [e_A] := \mathsf{proc}\ \mathcal{F}([t|\_]^*)
\end{array}
$$

Fig. 2. Abstract syntax for program statements.

simplest way to enable programs to write non-constant procedures onto the heap. As our syntax uses _ to represent arguments not yet "filled in", we can supply any subset of the arguments, not just initial segments. The statement $\mathsf{eval}[a](\boldsymbol{t})$ runs the procedure stored on the heap at address a, with value parameters $\boldsymbol{t}$, faulting if address a does not contain a procedure of the appropriate arity.

2.2 Assertion language

Fig. 3 gives the syntax for the assertion language. Based on [17], the language allows *nested triples* to appear in assertions, such that we can reason about stored procedures. The assertion $x \mapsto \forall a.\, \{a \mapsto \_\} \cdot (a)\, \{a \mapsto \_\}$, for example, states that the content at address x is a procedure which satisfies the given Hoare triple.[2] Additions to the logic of [17] are the set and element expressions. In the formula $\mathrm{P}({e_V}^*; {e_S}^*)$, the ; separates integer arguments from set arguments. An assertion is called *pure* if it is made up only of (in)equalities, set constraints and predicates whose definitions are pure; pure formulae do not depend on the heap[3].

When building formal verification tools there is a trade-off between expressiveness of the specifications that one considers, and the degree of automation one can achieve. Rather than using the full assertion language, we restrict ourselves to the fragment given in Fig. 3; in return for this sacrifice we are able to program an effective automatic entailment prover in a fairly natural way.

2.3 Crowfoot input language

Crowfoot accepts annotated programs written using the programming and assertion languages given in the previous subsections. Specifically, a Crowfoot input

[2] During the proof process, constants may be substituted into the arguments of the nested triple, which explains why the definition of B in Fig. 3 uses t.

[3] Here for convenience we follow Smallfoot's implementation and have only one kind of conjunction $\star$ in our logic; we do not include $\wedge$. The pure formulae such as $x = y$ are then given a non-standard interpretation, also requiring that the heap be empty.

set variables α, predicate names P

element expressions	e_E	::=	$e_V \mid (e_E{}^+)$
set expressions	e_S	::=	$\alpha \mid e_S \cup e_S \mid \{e_E\} \mid \emptyset$
behavioural spec.	B	::=	$\forall [x\|\alpha]^*.\ \{P\} \cdot (t^*)\{Q\}$
content spec.	$\mathscr{C}$	::=	$e_V \mid \_ \mid B$
atomic formula	A	::=	$e_A \mapsto \mathscr{C}^+ \mid \mathrm{P}(e_V{}^*; e_S{}^*) \mid e_V = e_V \mid e_V \neq e_V$
			$\mid e_E \in e_S \mid e_E \notin e_S \mid e_S \subseteq e_S \mid e_S = e_S$
spatial conjunction	Φ, Θ, Υ	::=	$\mathsf{emp} \mid A \star \Theta$
assertion disjunct	Ψ	::=	$\exists [x\|\alpha]^*.\Theta$
assertion	P, Q	::=	$\mathit{false} \mid \Psi \ \vee \ P$

Fig. 3. Abstract syntax for Crowfoot's assertion language.

program is a sequence of declarations, which can be of the following kinds:

$$
\begin{aligned}
\text{decl} ::= \ & \mathsf{const}\ c \mid \mathsf{const}\ c = n \mid \mathsf{forall}\ \mathrm{P} \\
& \mid \mathsf{recdef}\ \mathrm{P}(x^*; \alpha^*) := P \mid \mathsf{recdef}\ \mathrm{P}(x^*) := \mathrm{P}(x^*) \circ \Psi \\
& \mid \mathsf{proc}\ \mathcal{F}(x^*)\ \mathsf{forall}\ [x|\alpha]^*.\ \mathsf{pre} : P\ \mathsf{post} : Q\ \{\ \mathsf{locals}\ x^*;\ C\ \} \\
& \mid \mathsf{proc}\ \mathsf{abstract}\ \mathcal{F}(x^*)\ \mathsf{forall}\ [x|\alpha]^*.\ \mathsf{pre} : P\ \mathsf{post} : Q
\end{aligned}
$$

The keyword const is used to declare named constants, optionally with a particular value. The keyword recdef is used to declare user-defined inductive or recursive predicates, such as for linked data structures and for recursion through the store. Examples, in Fig. 4, will be discussed in the next section. Declaration forall P declares P to be an "abstract" or universally quantified predicate, i.e. one that may be used in specifications but has no definition (and thus cannot be folded or unfolded).

Finally, the keyword proc is used to declare procedures. Procedures have a name, a formal parameter list, a pre- and post-condition, and a body. The forall keyword is used to universally quantify variables over both the pre- and post-condition. Procedures declared as abstract do not have a body, just a specification; abstract procedures are typically used when we want to describe the behaviour of some library routine without giving an implementation.

Statement annotations In programs checked by Crowfoot, some of the statements need to be annotated with extra information to help the verifier. These annotations consist of the following changes to the statement grammar of Fig. 2:

$$
\begin{aligned}
\text{statement } C \quad ::= \quad & \ldots \mid \mathsf{while}\ e_V\ [=|\neq]\ e_V\ \boxed{P}\ \mathsf{do}\ C \\
\text{atomicst } At \quad ::= \quad & \ldots \mid \boxed{\mathsf{ghost}\ \text{ghoststmt}} \mid \mathsf{call}\ \mathcal{F}(t^*)\ \boxed{\text{deepfr-annot?}} \\
& \mid [e_A] := \mathsf{proc}\ \mathcal{F}([t|\_]^*)\ \boxed{\text{deepfr-annot?}} \\
\text{ghoststmt} \quad ::= \quad & \mathsf{fold}\ \mathrm{P}([e_V|?]^*; [e_S|?]^*) \mid \mathsf{unfold}\ \mathrm{P}([e_V|?]^*; [e_S|?]^*) \\
\text{deepfr-annot} \quad ::= \quad & \mathsf{deepframe}\ \Psi
\end{aligned}
$$

Loops are annotated with invariants (as in Smallfoot and VeriFast). Like VeriFast, Crowfoot needs annotations to indicate at which locations it is necessary to fold or unfold user-defined predicates[4]. These annotations take the form of ghost fold and ghost unfold statements. For example, in order to reason about code which disposes the head of a linked list, one needs to unfold the inductively defined list predicate to expose the head node. Arguments to predicates being folded and unfolded can be given, or they can be left blank using '?' in which case Crowfoot attempts to find appropriate instantiations. Crowfoot is able to recognise predicate definitions which fit a general pattern for being "list-segment-like", and two further ghost statements split and join are available for these; as they are not needed in our running example we will not describe them.

2.4 Deep framing

The *deep frame rule* [5, 17] allows one to infer $\{P\}\,C\,\{Q\} \otimes I$ from $\{P\}\,C\,\{Q\}$, where $\otimes$ is a deep framing operator. Intuitively this operator adds the invariant I not just to the pre- and post-conditions of the triple $\{P\}\,C\,\{Q\}$, but also to all triples nested *inside* P and Q, at all levels. For example,

$$\begin{aligned} & \forall a.\, \{a \mapsto \{\mathsf{emp}\} \cdot ()\, \{\mathsf{emp}\}\} \cdot (a)\, \{\mathsf{emp}\} \otimes y \mapsto \_ \\ \Leftrightarrow \quad & \forall a.\, \{a \mapsto \{y \mapsto \_\} \cdot ()\, \{y \mapsto \_\} \star y \mapsto \_\} \cdot (a)\, \{y \mapsto \_\} \end{aligned}$$

as can be proved using the distribution laws for $\otimes$ found in [17]. This is useful for modular reasoning as explained in [5] and as will be demonstrated by our running example. The operator $\circ$ from [17], used in recdef definitions, is a convenient shorthand: $A \circ I := (A \otimes I) \star I$.

The annotation deepframe I tells Crowfoot to add the invariant I deeply onto the triple for a procedure; this can be done when a procedure is called with call, or when a procedure is first written to the heap (but not with eval [8]).

Crowfoot implements deep framing using the $\otimes$ distribution laws from [17]. However there is no simple law for distributing $\otimes$ through recursively defined predicates; instead, Crowfoot uses the following lemma.

Lemma 1. *Given the following predicate definition*

$$\mathrm{R}(\boldsymbol{x}) \quad := \quad \bigstar_{i=1}^{n} v_i \mapsto \forall \boldsymbol{a}_i.\, \{\mathrm{R}(\boldsymbol{e}) \;\star\; F_i\} \cdot (\boldsymbol{p}_i)\, \{\mathrm{R}(\boldsymbol{e}) \;\star\; G_i\} \;\star\; H$$

where: $\boldsymbol{e}$ may contain variables $\boldsymbol{a}_i$ as well as $\boldsymbol{x}$, each F_i, each G_i and H are all left zeroes of $\otimes$ (i.e. informally they do not contain any nested triples), let us define $\mathrm{S}(\boldsymbol{x}, \boldsymbol{y}) := \mathrm{R}(\boldsymbol{x}) \circ T(\boldsymbol{y})$ where $fv(T(\boldsymbol{y})) = \boldsymbol{y}$ and $\boldsymbol{y} \cap \boldsymbol{a}_i = \emptyset$ (implicitly also $\boldsymbol{x} \cap \boldsymbol{y} = \emptyset$). Note that T may contain occurrences of S again. Then the following equivalence holds:

$$\mathrm{S}(\boldsymbol{x}, \boldsymbol{y}) \;\Leftrightarrow\; \left(\bigstar_{i=1}^{n} v_i \mapsto \forall \boldsymbol{a}_i.\, \{\mathrm{S}(\boldsymbol{e}, \boldsymbol{y}) \;\star\; F_i\} \cdot (\boldsymbol{p}_i)\, \{\mathrm{S}(\boldsymbol{e}, \boldsymbol{y}) \;\star\; G_i\} \right) \;\star\; H \;\star\; T(\boldsymbol{y})$$

[4] Smallfoot [3] did not need fold/unfold ghost statements because only particular built-in list and tree predicates were available. Crowfoot allows users to write their own inductive definitions and thus, like VeriFast[13], requires extra annotations.

$$\mathsf{recdef}\ \$Rel(n,m) := \quad n \leq 0 \star m = 0 \quad \vee \quad n = 1 \star m = 1 \\ \vee \quad \exists a, b.\ 2 \leq n \star \$Rel(n-2,a) \star \$Rel(n-1,b) \star m = a + b$$

$$\mathsf{recdef}\ \$RecFn(f) := f \mapsto \forall n, a. \\ \{\$RecFn(a) \star res \mapsto \_\} \cdot (a,n)\ \{\exists v.\ \$RecFn(a) \star res \mapsto v * \$Rel(n,v)\}$$

$$\mathsf{recdef}\ \$ListLibStrong(lookupL, addL, createL, disposeL) := \\ lookupL \mapsto \ldots \star createL \mapsto \ldots \star disposeL \mapsto \ldots \\ \star\ addL \mapsto \forall al, key, value, \kappa. \\ \left\{ \begin{array}{l} \$AssocListH(al;\kappa) \\ \star\ \$Rel(key,value) \end{array} \right\} \cdot (al, key, value)\ \ \left\{ \$AssocListH(al; \{key\} \cup \kappa) \right\}$$

$$\mathsf{recdef}\ \$ListLibWeak(lookupL, addL, createL, disposeL) := \\ lookupL \mapsto \ldots \star createL \mapsto \ldots \star disposeL \mapsto \ldots \\ \star\ addL \mapsto \forall al, key, value. \\ \left\{ \begin{array}{l} \exists \kappa.\ \$AssocListH(al;\kappa) \\ \star\ \$Rel(key,value) \end{array} \right\} \cdot (al, key, value)\ \ \left\{ \begin{array}{l} \exists \kappa. \\ \$AssocListH(al;\kappa) \end{array} \right\}$$

$$\mathsf{recdef}\ \$S(a, f, al, lookupL, addL, createL, disposeL)\ :=\ \$RecFn(a) \circ DeepInv$$

where $DeepInv$ abbreviates

$$\exists \kappa. \left(\begin{array}{l} f \mapsto \forall n, a. \\ \qquad \left\{ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto \_ \right\} \\ \qquad\qquad \cdot(a,n) \\ \qquad \left\{ \exists v.\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto v \star \$Rel(n,v) \right\} \\ \star\ \$AssocListH(al;\kappa) \star \$ListLibWeak(lookupL, addL, createL, disposeL) \end{array} \right)$$

$$\mathsf{recdef}\ \$AssocList(x;\tau) \quad := \quad x = 0 \star \tau = \emptyset \\ \vee\ \exists next, k, v, \tau'.\ x \mapsto k, v, next \star \$Rel(k,v) \star \$AssocList(next;\tau') \star \tau = \{k\} \cup \tau'$$

$$\mathsf{recdef}\ \$AssocListH(x;\tau) := \exists y.\ x \mapsto y \star \$AssocList(y;\tau)$$

Fig. 4. User-defined predicates used to specify and verify our running example.

3 Specification of the running example

The specifications of the procedures in Fig. 1 can be found in Fig. 5. The auxiliary predicate definitions are given in Fig. 4.

The *fib* implementation. Let us first examine how to specify the *fib* code. Predicate $\$Rel(n,m)$ says that n and m are appropriately related for the function

proc $main()$
  pre : $res \mapsto \_$;
  post : emp;

proc $fib(a, n)$
  pre : $\$RecFn(a) \star res \mapsto \_$;
  post : $\exists v.\ \$RecFn(a) \star res \mapsto v \star \$Rel(n, v)$;

proc $mem(lookupL, addL, createL, disposeL, al, f, a, n)$
  pre : $\$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto \_$;
  post : $\exists m.\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto m \star \$Rel(n, m)$;

proc $useFib(lookupL, addL, createL, disposeL)$
  pre : $res \mapsto \_ \star \$ListLibWeak(lookupL, addL, createL, disposeL)$; post : emp;

proc $load\_list\_lib(lookupL, addL, createL, disposeL)$
  pre : $lookupL \mapsto \_ \star addL \mapsto \_ \star createL \mapsto \_ \star disposeL \mapsto \_$;
  post : $\$ListLibStrong(lookupL, addL, createL, disposeL)$;

Fig. 5. Procedure specifications for the memoiser example.

being computed; in this case we define $\$Rel(n, m)$ to mean that m is the nth Fibonacci number. But this definition is only used inside the proof of *fib*, and not when proving the generic components such as *mem*.

Suppose we try to write a precondition for the *fib* code. This precondition must mention all the heap resources needed by *fib*. Firstly a cell $res \mapsto \_$ is needed into which we write the result. Secondly, since *fib* makes its recursive call through the heap at the address given by parameter a, the precondition must include $a \mapsto B$ where B is a nested triple. In particular, B must state that the code stored at a has the same kind of behaviour as we specify for the *fib* procedure. But we don't have *fib*'s specification yet, because we are still trying to formulate its precondition! It appears that we need a specification which depends on itself. Using the recdef keyword we can declare such a recursively defined specification, namely the $\$RecFn$ predicate, which appears nested inside its own definition.

The memoiser. The memoiser implementation uses an association list data structure, at address *al*, to cache the input-output pairs for the function being memoised. An association list with a header cell, starting at address *al* and containing values for a set κ of keys, is described by $\$AssocListH(al; \kappa)$. Such lists are manipulated via four library routines, pointers to which are passed in the arguments *lookupL*, *addL*, *createL*, *disposeL*. Argument f to procedure *mem* is a pointer to the code of the function being memoised; the memoiser must call this code when the required data is not found in the cache. The arguments *lookupL*, *addL*, *createL*, *disposeL*, al, f are fixed by partial application when the memoiser is first loaded onto the heap. This leaves a two-argument procedure: the first argument a is passed straight through to the function being memoised, and the second argument n is the input at which to apply the function.

The memoiser is designed to be placed into mutual recursion with *fib*, or similar code for computing other functions. During computations the *fib* code and the memoiser then invoke each other in a "zig-zag" mutual recursion. The

"ensemble" of these two functions stored on the heap and able to invoke each other can be described by $\$S(a, f, al, lookupL, addL, createL, disposeL)$ which, by Lemma 1, is equivalent to:

$$\exists\kappa.\quad a \mapsto RecFnMem(\cdot) \star f \mapsto RecFnMem(\cdot) \\ \star \$AssocListH(al;\kappa) \ \star \ \$ListLibWeak(lookupL, addL, createL, disposeL)$$

where $RecFnMem(\cdot)$ is shorthand for

$$\forall a, n. \quad \big\{\, \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto \_ \,\big\} \\ \cdot(a, n) \\ \big\{\, \exists v.\ \$S(a, f, al, lookupL, addL, createL, disposeL) \star res \mapsto v \star \$Rel(n, v) \,\big\}$$

Intuitively *RecFnMem* describes code which computes a function as specified by $\$Rel$, provided the heap contains the "ensemble" of function and memoiser code as described above.

The main program. The *main* procedure first calls *load_list_lib* to load the association list library routines onto the heap. Then, *main* invokes *useFib* which loads the *fib* code and the memoiser, places them into mutual recursion, and finally uses this to compute the 31337th Fibonacci number.

In *useFib* we see the crucial role of the deep frame rule. We have specified (and Crowfoot will prove) *fib* for the case where it is placed in recursion *only with itself*, using $\$RecFn$. Hence, if the deepframe annotation were not used in *useFib*, the symbolic heap after the statement $[f] :=$ proc *fib*(_, _) would contain

$$f \mapsto \forall a, n. \{\$RecFn(a) \star res \mapsto \_\}\cdot(a, n)\,\{\exists v.\ \$RecFn(a) \star res \mapsto v \star \$Rel(n, v)\}$$

However the annotation deepframe *DeepInv* tells Crowfoot to apply $-\otimes DeepInv$ to the above triple, resulting in $RecFnMem(\cdot)$. In this way, we have used the deep frame rule to *derive* another specification for the *fib* code, which describes how that code works in mutual recursion with a memoiser. We did not need to respecify or reprove *fib*.

The list library. The memoiser depends only on relatively weak properties of the association list library; a library with these properties is specified by $\$ListLibWeak$. But the list library is specified with a stronger specification $\$ListLibStrong$ so that it can also be used with other clients which need additional guarantees. Specifications for three of the routines are omitted in Fig. 4, but with the remaining "add" routine one can see a difference. In order to compute the correct function, the memoiser does not care whether the $(key, value)$ pair is actually added to the list or not, as long as whatever pairs are in the list afterwards are suitably related by $\$Rel$. But other clients of the list library will certainly care about this.

Our verification will go through because Crowfoot can prove the entailment

$$\$ListLibStrong(lookupL, addL, createL, disposeL) \\ \Rightarrow\ \$ListLibWeak(lookupL, addL, createL, disposeL) \qquad (1)$$

as we shall discuss in Section 5. Having such entailments proved automatically facilitates reasoning when one is "plugging together" different pieces of code.

4 Automation of program verification

4.1 Overview

The introduction of nested triples into the logic increases considerably the difficulty of proving entailments automatically: because assertions can contain triples and vice versa, we need provers for entailments between both assertions and triples, and these provers need to invoke each other. In fact, at the heart of Crowfoot are automated provers for five interrelated judgements:

- Symbolic execution: $\Pi, \Gamma \triangleright \{P\}C\{\ Q\ \}$
- Entailment between assertion disjuncts: $\Phi \vdash^{I} \exists \boldsymbol{v}.\, \Upsilon \star \Theta$
- Entailment between behavioural specifications (triples): $B_1 \vdash B_2$
- Computing the postcondition for a call or eval: $B \vdash_{\text{find-post}} \{\Phi\} \cdot (\boldsymbol{t})\{\ Q\ \}$
- Finding specifications inside a symbolic state: $\Upsilon \vdash_{\text{find-tr}} e \mapsto B$

Here, Π is a *predicate context* mapping predicate names to their definitions (given by recdef) and Γ is a *procedure context* mapping fixed procedure names to their specifications (given by pre and post). C is a program statement. (As in Fig. 3) P, Q are assertions, Ψ is used for assertion disjuncts, and Φ, Θ, Υ for spatial conjunctions. Behavioural specifications are named B, and I is an *instantiation map* mapping the existentially quantified variables $\boldsymbol{v}$ to appropriate witnesses.

Shaded variables (such as the frame Θ) are those whose value is not given as an input to the prover, but rather is inferred by the proof rules. The meanings of these judgements can be seen in the following soundness theorem, which for now we simply state; we will discuss it later in Section 4.4.

Theorem 1. Soundness theorem. *Our five proof systems are sound, that is:*

- *If* $\Phi \vdash^{I} \exists \boldsymbol{v}.\Upsilon \star \Theta$ *(where* $fv(\Phi) \cap \boldsymbol{v} = \emptyset$*) then* $\Phi \Rightarrow \Upsilon[\boldsymbol{v} \backslash I(\boldsymbol{v})] \star \Theta$ *where:* $fv(\Theta) \subseteq fv(\Phi)$*,* $dom(I) = \boldsymbol{v}$ *and* $fv(Im(I)) \subseteq fv(\Phi)$*.*
- *If* $B_1 \vdash B_2$ *then* $B_1 \Rightarrow B_2$*.*
- *If* $B \vdash_{\mathit{find\text{-}post}} \{\Phi\} \cdot (\boldsymbol{t})\, \{Q\}$ *then* $B \Rightarrow \{\Phi\} \cdot (\boldsymbol{t})\, \{Q\}$*.*
- *If* $\Upsilon \vdash_{\mathit{find\text{-}tr}} e \mapsto B$ *then* $\Upsilon \Leftrightarrow e \mapsto B \star \Upsilon'$ *for some* Υ'*.*
- *Our symbolic execution rules are sound.* □

Verification of a program by Crowfoot proceeds as follows. First, Crowfoot's verification condition (VC) generator reads in the annotated program and produces a set of VCs, each of the form $\Pi; \Gamma \triangleright \{P\}\, C\, \{Q\}$, such that if all the VCs hold then the input program meets its specifications. There is one such VC for each concrete procedure of the input program, with C being its body and P, Q the pre- and post-conditions. Then the generated VCs are passed to the symbolic execution engine which attempts to prove them. During symbolic execution, entailment problems of various kinds arise: for instance when the end of a procedure (resp. loop body) is reached, one must check an entailment between the current symbolic state and the postcondition (resp. loop invariant). These entailments

(between assertions) may give rise to entailments between triples because triples can appear nested. When an eval statement is reached, $\vdash_{\text{find-tr}}$ is employed to find a triple to use for the invocation, and then $\vdash_{\text{find-post}}$ is used to compute a symbolic state holding after the invoked code returns.

4.2 Symbolic execution engine

Crowfoot's symbolic execution engine is based on ideas put forward in [4] and now well established. The symbolic execution rules, a few of which can be seen in Fig. 7, depend on all four of the other judgements. One such rule is:

Lookup

$$\frac{\mathit{purify}(\Upsilon) \vdash_{SMT} E = G + o \qquad \Pi; \Gamma \triangleright \{x = (e[x\backslash x']) \ \star (\Upsilon \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n)[x\backslash x']\}\, C\, \{Q\}}{\Pi; \Gamma \triangleright \{\Upsilon \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, e, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\}\, x := [E];\ C\, \{Q\}}\ x' \text{ fresh}$$

where $\mathit{purify}(\Upsilon)$ extracts the pure parts of Υ, and $\vdash_{SMT}$ represents sending a pure goal to an SMT solver to be checked. The rules which are intrinsically new in our work are those for the statements which make use of higher-order store, namely eval $[E](\boldsymbol{t})$ and $[E] :=$ proc $\mathcal{F}([t|\_]^*)$. The rule for eval (where $\boldsymbol{a} \in [x|\alpha]^*$) is:

Eval

$$\frac{\begin{array}{c}\Upsilon \vdash_{\text{find-tr}} E \mapsto \forall \boldsymbol{a}.\{P\} \cdot (\boldsymbol{t})\{Q\} \\ \forall \boldsymbol{a}.\{P\} \cdot (\boldsymbol{t})\{Q\} \vdash_{\text{find-post}} \{\Upsilon\} \cdot (\boldsymbol{t'}) \left\{ \bigvee_{i=1}^{m} \exists \boldsymbol{v}_i.\Phi_i \right\} \\ \Pi; \Gamma \triangleright \left\{ \bigvee_{i=1}^{m} \Phi_i[\boldsymbol{v_i}\backslash \boldsymbol{v_i'}] \right\} C\, \{Q'\}\end{array}}{\Pi; \Gamma \triangleright \{\Upsilon\}\ \mathsf{eval}\ [E](\boldsymbol{t'})\ ;\ C\, \{Q'\}}\ \boldsymbol{v_i'} \text{ fresh}$$

This uses the $\vdash_{\text{find-tr}}$ prover to find the specification $\forall \boldsymbol{a}.\{P\} \cdot (\boldsymbol{t})\{Q\}$ of the code being invoked from the heap. Then the $\vdash_{\text{find-post}}$ prover is used to compute all the possible symbolic states $\exists \boldsymbol{v}_i.\Phi_i$ that may result from running that code. Finally, symbolic execution is performed on the "continuation" statement C.

4.3 Entailment provers

We sketch how our different entailment provers work. The selected rules we refer to are listed in Fig. 6.

Entailments between assertion disjuncts. The main part of these proofs involves successively cancelling spatial formulae from the left and right sides of $\vdash$. Sometimes these steps involve computing witnesses for existentially quantified variables, which are added to the instantiation map I. For instance, the goal

$$\Phi \star x \mapsto 3 \vdash^{I} \exists u, \boldsymbol{v}.\ \Upsilon \star x \mapsto u \star \Theta \quad \text{reduces to} \quad \Phi \vdash^{I'} \exists \boldsymbol{v}.\ \Upsilon[u\backslash 3] \star \Theta$$

by CancelPtInstContents, where we will take $I := I'[u := 3]$. Note how the rule CancelPtTriple for cancelling cells containing code invokes the prover for entailments between triples (specifications). The cancellation rules are designed

to reduce the goal to the form $\Upsilon \vdash^I \Phi \star \Theta$ where Φ is pure. We finish by sending the pure entailment problem $\mathit{purify}(\Upsilon) \vdash_{SMT} \Phi$ to an SMT solver, and we take Υ as the inferred frame Θ.

Entailments between specifications. Most of the work of proving judgements $B_1 \vdash B_2$ is done by the TripleEnt rule, which breaks down the checking of an entailment $B \vdash \{\Phi\} \cdot (\boldsymbol{t}) \{Q'\}$ between specifications into two tasks. Intuitively, we first use $\vdash_{\text{find-post}}$ to try to compute a state Q we will end up in if we run some code with specification B in a state satisfying Φ. We then check whether Q implies the postcondition Q'.

Inferring postconditions for invocations. The main rule for $\vdash_{\text{find-post}}$ is InferSpecForCall. Underlying it is a combination of $\forall$-instantiation, the shallow frame axiom $\{P\}C\{Q\} \Rightarrow \{P \star R\}C\{Q \star R\}$ and the consequence axiom.

Finding specifications inside a symbolic state. To be able to symbolically execute an $\mathsf{eval}\ [e](\boldsymbol{t})$ statement, we need to first find in our symbolic heap a cell $e \mapsto B$; we can then use the specification B to reason about the invocation. We use $\vdash_{\text{find-tr}}$ for finding such specifications. The most commonly used proof rule for $\vdash_{\text{find-tr}}$ is Find which covers the case when the required cell $e \mapsto B$ is already explicitly present in the symbolic heap. Other proof rules, omitted for space reasons, look inside occurrences of user-defined predicates to find the appropriate specification.

4.4 Theoretical basis

One distinctive feature of our tool is that we can prove its soundness, embodied by Theorem 1. Due to lack of space we cannot go into detail, but we briefly explain our soundness argument. Soundness is proved with respect to another logic with nested triples, an extension of the logic of [17][5] which in turn has been proved sound in *loc. cit.* via a model construction. It is relatively straightforward to construct a step-indexed analogue which encompasses Crowfoot's extra features. It should be pointed out that soundness only holds for recursive predicates that *exist*. For a predicate R such as $\$\mathit{RecFn}$ (Fig. 4), existence is guaranteed because in its definition, R itself always occurs inside pre- and postconditions of some triple. The corresponding semantic functional is contractive and thus the predicate does exist via Banach's fixpoint theorem. In general however, reasons for existence may not be immediately clear, particularly for definitions that combine the recursive uses of R (as in $\$\mathit{RecFn}$) with inductive uses of R, as found in definitions of linked list predicates. Our tool does not check for existence.

5 Excerpt from the verification of the running example

During the symbolic execution of *main*, we see how the entailment prover for assertions and the prover for specifications are mutually recursive. Before calling

[5] enriched by inductive and abstract predicates as well as recursively defined procedures with explicit calls

InferSpecForCall

$$\frac{\Phi \;\vdash^{I}\; \exists \boldsymbol{u}_k, \boldsymbol{a}.\, \Upsilon_k \star \Theta}{\forall \boldsymbol{a}. \left\{ \bigvee_{i=1}^{n} \exists \boldsymbol{u}_i.\Upsilon_i \right\} \cdot (\boldsymbol{t}) \left\{ \bigvee_{i=1}^{m} \exists \boldsymbol{v}_i.\Upsilon'_i \right\} \;\vdash_{\text{find-post}}\; \{\Phi\} \cdot (\boldsymbol{t}) \left\{ \bigvee_{i=1}^{m} (\exists \boldsymbol{v}_i.\Upsilon'_i[\boldsymbol{a} \backslash I(\boldsymbol{a})] \star \Theta) \right\}}$$

1. $\boldsymbol{t} \cap \boldsymbol{a} = \emptyset$
2. $fv(\Phi) \cap \boldsymbol{u}_k = \emptyset$ and $fv(\Phi) \cap \boldsymbol{a} = \emptyset$
3. for each $i \in \{1, \ldots, m\}$ we have $\boldsymbol{v}_i \cap \boldsymbol{a} = \emptyset$
4. for each $i \in \{1, \ldots, m\}$, no formula in $I(\boldsymbol{a})$ contains a variable from $\boldsymbol{v}_i$
5. $k \in \{1, \ldots, n\}$
6. $\boldsymbol{u}_k \cap \boldsymbol{a} = \emptyset$

TripleEnt

$$\frac{B \;\vdash_{\text{find-post}}\; \{\Phi\} \cdot (\boldsymbol{t}) \left\{ \bigvee_{i=1}^{m} \exists \boldsymbol{v}_i.\Upsilon_i \right\} \qquad \bigwedge_{i=1}^{m} \left(\Upsilon_i[\boldsymbol{v}_i \backslash \boldsymbol{a}_i] \;\vdash^{I_i}\; \exists \boldsymbol{b}_{j_i}.(\Upsilon'_{j_i}[\boldsymbol{w}_{j_i} \backslash \boldsymbol{b}_{j_i}]) \star \Theta_i \right)}{B \;\vdash\; \{\Phi\} \cdot (\boldsymbol{t}) \left\{ \bigvee_{i=1}^{m'} \exists \boldsymbol{w}_i.\Upsilon'_i \right\}}$$

1. $j_1, \ldots, j_m \in \{1, \ldots, m'\}$
2. $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$ all chosen fresh
3. $\boldsymbol{b}_{j_1}, \ldots, \boldsymbol{b}_{j_{m'}}$ all chosen fresh
4. $\Theta_1, \ldots, \Theta_m$ pure

CancelPtInstContents

$$\frac{\Phi \;\vdash^{I}\; \exists \boldsymbol{v}\;.\; \Upsilon[v \backslash E] \star \Theta}{\Phi \star e \mapsto E \;\vdash^{I[v:=E]}\; \exists \boldsymbol{v}, v\;.\; \Upsilon \star e' \mapsto v \star \Theta}$$

1. $fv(e') \cap \boldsymbol{v} = \emptyset$
2. $v \notin fv(e')$
3. $purify(\Phi) \vdash_{SMT} e = e'$

CancelPtTriple

$$\frac{\Phi \;\vdash^{I}\; \exists \boldsymbol{v}\;.\; \Upsilon \star \Theta \qquad B_1 \vdash B_2}{\Phi \star e \mapsto B_1 \;\vdash^{I}\; \exists \boldsymbol{v}\;.\; \Upsilon \star e' \mapsto B_2 \star \Theta}$$

1. $fv(e', B_2) \cap \boldsymbol{v} = \emptyset$
2. $purify(\Phi) \vdash_{SMT} e = e'$

Find

$$\frac{}{\Phi \star E \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, B, \mathscr{C}_{o+1}, \ldots \mathscr{C}_n \vdash_{\text{find-tr}} e \mapsto B} \; purify(\Phi) \vdash_{SMT} e = E + o$$

Fig. 6. Notable rules used in our automatic entailment provers.

the *useFib* procedure, the $\$ListLibStrong$ predicate is unfolded, and folded up into $\$ListLibWeak$. This essentially means proving (1) on page 9, which is an entailment between assertion disjuncts.

The proof proceeds by cancelling out the atomic formulae, which in this case means using CancelPtTriple for each of the four library procedures. This is where the entailment prover for specifications is needed: the premise of this rule requires that each strong specification entails the respective weak variation.

This entailment is checked by the TripleEnt rule, which has two premises. The first uses the judgement $\vdash_{\text{find-post}}$ (with InferSpecForCall) which will check that the weak pre-condition entails the strong pre-condition, with some inferred frame left over (in this case the frame is trivial). For the second premise it is

required to prove that the strong postcondition (together with the frame) entails the weak one. Using again the entailment prover for assertion disjuncts, Crowfoot proves: $\$AssocListH(al, \{key\} \cup \kappa) \vdash^{[\kappa' \mapsto \{key\} \cup \kappa]} \exists \kappa'.\ \$AssocListH(al, \kappa')$.

6 Related and future work

Crowfoot can be considered as extending Smallfoot [4] (though Crowfoot was written from scratch) by allowing (partially applicable) procedures to be stored on the heap. Our assertion language uses nested triples to specify stored procedures and recursively defined assertions to deal with recursion through the store. Crowfoot uses an SMT solver to deal with pure assertions and therefore can be used to prove more than just memory safety (see our example).

The system most closely related to Crowfoot is the VeriFast [13, 12] tool, also based on symbolic execution with separation logic. VeriFast supports a C-like language (and also Java) and supports C-style function pointers. Functions in the C-like language live in an immutable memory and can be pointed to but not updated, whereas Crowfoot's programming language stores procedures in dynamic, mutable memory. However these setups seem to have a similar character.

A key difference is that while Crowfoot uses nested triples to express requirements for procedure pointers, VeriFast expresses such requirements via *function types* with which the C type system is extended. A function type declaration associates a pre- and post-condition with the function type; the declared type can have extra arguments to simulate nested triples which can contain free variables. These can be recursive since for every function type F there is a predicate 'is_F(_)' which states that (the function pointed to by) its first argument satisfies the "contract" for function type F (possibly with additional arguments).

Crowfoot offers some features which VeriFast does not, such as partial application of which our example makes essential use in *useFib* when loading the memoiser *mem*. Another important feature to support stored procedures is entailment between Hoare triples which is automated in our verifier and needed in our example, as explained in Section 5. VeriFast does not support such proofs (which in that system would be proofs of entailments of shape is_F(_) $\Rightarrow$ is_G(_)), even manual ones, whereas Crowfoot finds them automatically. Crowfoot supports annotations for deep frame rule application (thus implementing the $\otimes$ operator) and allows extensions of predicates via $\circ$, thus allowing elegant use of deep framing on recursively defined specifications (cf. definition of $\$S$ in our example in Fig. 4). In VeriFast one can simulate the effect of the deep frame rule by using (second order) function types which take as argument a predicate representing the deeply framed invariant. However, this means one must write all specifications that can appear for stored procedures *a priori* in that style.

On the other hand, VeriFast offers features that Crowfoot does not, such as concurrency, termination checking and the use of more types (such as mathematical lists and functions on them) in the assertions. VeriFast's support for second order logic is useful for specifying and reasoning about higher-order and polymorphic functions.

Other related work includes four systems developed in Coq: XCAP [16], Bedrock [10], GCAP [6] and Ynot [14]. XCAP allows reasoning about programs which use pointers to (immutable) functions, by introducing a special cptr predicate which in proofs behaves much like nested triples, though its underlying semantics is very different. GCAP is a related system supporting reasoning about low-level runtime code modification. Ynot builds a type theory in which Hoare triples ("Hoare types") can be used as the types for side-effecting commands; these Hoare types can be nested. To our knowledge, Ynot does not support recursion through the store.

Previous work [7] briefly described one application of Crowfoot, namely the verification of runtime code updates, but did not go into detail about Crowfoot, its implementation or its theoretical basis. An interactive version of Crowfoot, which includes the example of this paper and others, can be used online [1].

Future work. The following extensions would permit the verification of more examples. As the *antiframe rule* is consistent with the logic used in Crowfoot (as proved in [18]), annotations similar to those for the deep frame rule could be implemented to allow hiding of invariants in "antiframe style". Though we do not need it for deep framing like VeriFast does, second order logic would support the specification of parametric procedures. A minor but useful extension is to allow proper functions with result values. We believe that Lemma 1 can be generalised to support mutually recursive definitions and to allow deep framing onto abstract (universally quantified) predicates. We plan to investigate extensions required to support reasoning about reflective programs and, finally, it is likely that many fold/unfold annotations can be discovered automatically, as done in [15].

References

1. The Crowfoot website. `www.sussex.ac.uk/informatics/crowfoot`, 2011.
2. N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*, pages 301–312, 2009.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
4. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
5. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5), 2006.
6. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *PLDI*, pages 66–77, 2007.
7. N. Charlton, B. Horsfall, and B. Reus. Formal reasoning about runtime code update. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE Workshops*, pages 134–138. IEEE, 2011.
8. N. Charlton and B. Reus. A deeper understanding of the deep frame axiom. Extended abstract, presented at LOLA (Syntax and Semantics of Low Level Languages), 2010.

$$\textsc{New}\quad \frac{\Pi; \Gamma \triangleright \left\{\Phi[x\backslash x'] \star x \mapsto (e_0, \ldots, e_n)[x\backslash x']\right\} C \left\{Q\right\}}{\Pi; \Gamma \triangleright \left\{\Phi\right\} x := \mathsf{new}\ e_0, \ldots, e_n;\ C \left\{Q\right\}}\ x' \text{ fresh}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}\forall \boldsymbol{a}. \left\{P\right\} \cdot (\boldsymbol{t}) \left\{Q\right\}\ \otimes \Psi \vdash_{\text{find-post}} \left\{\Phi\right\} \cdot (\boldsymbol{t'}) \left\{\bigvee_{i=1}^{m} \exists \boldsymbol{v}_i. \Upsilon_i\right\} \\ \Pi; \forall \boldsymbol{a}. \left\{P\right\} \mathcal{F}(\boldsymbol{t}) \left\{Q\right\}, \Gamma \triangleright \left\{\bigvee_{i=1}^{m} \Upsilon_i[\boldsymbol{v_i}\backslash \boldsymbol{v_i'}]\right\} C \left\{Q'\right\}\end{array}}{\Pi; \forall \boldsymbol{a}. \left\{P\right\} \mathcal{F}(\boldsymbol{t}) \left\{Q\right\}, \Gamma \triangleright \left\{\Phi\right\} \mathsf{call}\ \mathcal{F}(\boldsymbol{t'})\ \mathsf{deepframe}\ \Psi;\ C \left\{Q'\right\}}\ \boldsymbol{v_i'} \text{ fresh}$$

$$\textsc{StoreCode}\quad \frac{\begin{array}{c}B = (\forall \boldsymbol{t}|_U, \boldsymbol{a}.\ \left\{P\right\} \cdot (\boldsymbol{t}|_U) \left\{Q\right\}) [\boldsymbol{t}|_{I\backslash U} \backslash \boldsymbol{r}|_{I\backslash U}] \\ \Pi; \Gamma, \forall \boldsymbol{t}, \boldsymbol{a}. \left\{P\right\} \mathcal{F}(\boldsymbol{t}) \left\{Q\right\} \triangleright \left\{\Phi \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_{o-1}, B \otimes \Psi, \mathscr{C}_{o+1}, \ldots, \mathscr{C}_n\right\} C \left\{Q'\right\}\end{array}}{\Pi; \Gamma, \forall \boldsymbol{t}, \boldsymbol{a}. \left\{P\right\} \mathcal{F}(\boldsymbol{t}) \left\{Q\right\} \triangleright \left\{\Phi \star G \mapsto \mathscr{C}_0, \ldots, \mathscr{C}_n\right\} [E] := \mathsf{proc}\ \mathcal{F}(\boldsymbol{r})\ \mathsf{deepframe}\ \Psi;\ C \left\{Q'\right\}}$$

1. $\boldsymbol{r} \in [x|c|\_]^*$
2. $\boldsymbol{a} = fv(P, Q) - \boldsymbol{t}$
3. $purify(\Phi) \vdash_{SMT} E = G + o$
4. $\boldsymbol{t} = (t_i)_{i \in I}$
5. $U = \{i \in I \mid r_i = \_\}$
6. $\boldsymbol{t}|_X = (t_i)_{i \in I \cap X}$

Fig. 7. Some of our symbolic execution rules.

9. N. Charlton and B. Reus. Specification patterns and proofs for recursion through the store. In *FCT*, pages 310–321, 2011.
10. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In M. W. Hall and D. A. Padua, editors, *PLDI*, pages 234–245. ACM, 2011.
11. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS*, pages 270–279, 2005.
12. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, pages 41–55, 2011.
13. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, pages 304–311, 2010.
14. A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.
15. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.
16. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *POPL*, pages 320–333, 2006.
17. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *CSL*, pages 440–454, 2009.
18. J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus. A semantic foundation for hidden state. In *FOSSACS*, pages 2–17, 2010.